Analysing and Compiling Coroutines with Abstract Conjunctive Partial Deduction

Danny De Schreye, Vincent Nys, and Colin Nicholson

Department of Computer Science, KU Leuven
Celestijnenlaan 200A, B-3001, Heverlee, Belgium

{danny.deschreye,vincent.nys}@cs.kuleuven.be

Abstract. We provide an approach to formally analyze the computational behavior of coroutines in Logic Programs and to compile these computations into new programs, not requiring any support for coroutines. The problem was already studied near to 30 years ago, in an analysis and transformation technique called Compiling Control. However, this technique had a strong ad hoc flavor: the completeness of the analysis was not well understood and its symbolic evaluation was also very ad hoc. We show how Abstract Conjunctive Partial Deduction, introduced by Leuschel in 2004, provides an appropriate setting to redefine Compiling Control. Leuschel's framework is more general than the original formulation, it is provably correct, and it can easily be applied for simple examples. We also show that the Abstract Conjunctive Partial Deduction framework needs some further extension to be able to deal with more complex examples.

1 Introduction

The work reported on in this paper is an initial step in a new project, in which we aim to formally analyze and automatically compile certain types of coroutining computations. Coroutines are a powerful means of supporting complex computation flows. They can be very useful for improving the efficiency of declaratively written programs, in particular for generate-and-test based programs. On the other hand, obtaining a deep understanding of the computation flows underlying the coroutines is notoriously difficult.

In this paper we restrict our attention to pure, definite Logic Programs. In this context, the problem was already studied nearly 30 years ago. Bruynooghe et al. [1986] and Bruynooghe et al. [1989] present an analysis and transformation technique for coroutines, called Compiling Control (CC for short). The purpose of the CC transformation is the following: transform a given program, P, into a program P', so that computation with P' under the standard selection rule mimics the computation with P under a non-standard selection rule. In particular, given a coroutining selection rule for a given Logic Program, the transformed program will execute the coroutining if it is evaluated under the standard selection rule of Prolog.

To achieve this, CC consists of two phases: an analysis phase and a synthesis phase. The analysis phase analyzes the computations of a program for a given query pattern and under a (non-standard) selection rule. The query pattern is expressed in terms of a combination of type, mode and aliasing information. The selection rule is instantiation-based, meaning that different choices in atom selection need to be based on different instantiations in these atoms. The analysis results in what is called a "trace tree", which is a finite upper part of a symbolic execution tree that one can construct for the given query pattern, selection rule and program. In the synthesis phase, a finite number of clauses are generated, so that each clause synthesizes the computation in some branch of the trace tree and such that all computations in the trace tree have been synthesized by some clause. The technique was implemented, formalized and proven correct, under certain fairly technical conditions.

Unfortunately, the CC transformation has a rather ad hoc flavor. It was very hard to show that the analysis phase of the transformation was complete, in the sense that a sufficiently large part of the computation had been analyzed to be able to capture all concrete computations that could possibly occur at run time. Even the very idea of a "symbolic execution" had an ad hoc flavor. It seemed that it should be possible to see this as an instance of a more general framework for analysis of computations.

Fortunately, since the development of CC a number of important advances have been achieved in analysis and transformation:

- General frameworks for abstract interpretation (e.g. Bruynooghe [1991]) were developed. It is clear that abstract interpretation has the potential to provide a better setting for developing the CC analysis. But it still seems different, because abstract interpretation is about analyzing properties that hold during or after a computation, while in CC we are interested in analyzing the computational behavior itself.
- Partial deduction of Logic Programs was developed (e.g. Gallagher [1986]). Partial deduction seems very similar to CC, but the exact relationship was never identified. When John Lloyd and John Shepherdson formalized the issues of correctness and completeness of partial deduction in Lloyd and Shepherdson [1991], this provided a new framework for thinking about a complete analysis of a computational behavior and it was clear that some variant of this could improve the CC analysis.
- Conjunctive partial deduction (see De Schreye et al. [1999]) seems even closer to CC. In an analysis for a CC transformation, one really does not want to split up the conjunctions of atoms into separate ones and then analyze the computations for these atoms separately. It is crucial that one can analyze the computation for certain atoms in conjunction (which is how conjunctive partial deduction generalizes partial deduction), so that their behavior under the non-standard selection rule may be observed.
- Finally, abstract (conjunctive) partial deduction (Leuschel [2004]) brings all these features together. It provides an extension of (conjunctive) partial

deduction in which the analysis is based on abstract interpretation, rather than on concrete evaluation.

In this paper we will demonstrate – mostly on the basis of examples – that abstract conjunctive partial deduction (ACPD for short) is indeed a suitable framework to redefine CC in such a way that the flaws of the original approach are overcome. We show that for simple problems in the CC context, ACPD can, in principle, produce the transformation automatically. We also show that for more complex CC transformations, ACPD is still not powerful enough. We suggest an extension to ACPD that allows us to solve the problem and illustrate with an example that this extension is very promising.

After the preliminaries, in Section 3, we introduce a fairly refined abstract domain, including type, mode and aliasing information, and we show, by means of an example, how ACPD allows us to analyze a coroutine and compile the transformed program. In Section 4 we propose a more complex example and show why it is out of scope for ACPD. We introduce an additional abstraction in our domain and illustrate that this abstraction solves the problem. This abstraction, however, does not respect the requirements of the formalization of ACPD in Leuschel [2004]. We end with a discussion.

2 Preliminaries

We assume that the reader is familiar with the basics of Logic Programming (Lloyd [1987]). We also assume knowledge of the basics of abstract interpretation (Bruynooghe [1991]) and of partial deduction (Lloyd and Shepherdson [1991]).

In this paper, names of variables will start with a capital. Names of constants will start with a lower case character. Given a program P, Con_p, Var_p, Fun_p and $Pred_p$ respectively denote the sets of all constants, variables, functors and predicate symbols in the language underlying P. $Term_p$ will denote the set of all terms constructable from Con_p, Var_p and Fun_p. $Atom_p$ denotes the set of all atoms which can be constructed from $Pred_p$ and $Term_p$. We will often need to refer to conjunctions of atoms of $Atom_p$ and we denote the set of all such conjunctions as $ConAtom_p$.

We will introduce an abstract domain in the following section. The abstract domain will be based on a set of abstract constant symbols, $ACon_p$. Based on these, there is a corresponding set of abstract terms, $ATerm_p$, which consists of the terms that can be constructed from $ACon_p$ and Fun_p. $AAtom_p$ will denote the set of abstract atoms, being the atoms which can be constructed from $ATerm_p$ and $Pred_p$. Finally, $AConAtom_p$ denotes the set of conjunctions of elements of $AAtom_p$.

3 An Example of a CC Transformation, Using ACPD

In this section, we provide the intuitions behind our approach by means of a simple example. We use permutation sort as an illustration. The intention is to transform this program so that calls to $perm/2$ and $ord/1$ are interleaved.

Example 1 (Permutation sort).

```
sort(X,Y) ← perm(X,Y), ord(Y).
perm([],[]).
perm([X|Y],[U|V]) ←
    del(U,[X|Y],W),perm(W,V).
del(X,[X|Y],Y).
del(X,[Y|U],[Y|V]) ← del(X,U,V).
ord([]).
ord([X]).
ord([X,Y|Z]) ←
    X ≤ Y, ord([Y|Z]).
```

We now introduce the abstract domain. This domain consists of two types of new constant symbols: g and $a_i, i \in \mathbb{N}$. The symbol g denotes any ground term in the concrete language. The basic intuition for the symbols a_i is that they are intended to represent variables of the concrete domain. However, as we want the abstract domain to be closed under substitution (if an abstract term denotes some concrete term, then it should also denote all of its instances), an abstract term a_i will actually represent any term of the concrete language.

The subscript i in a term a_i is used to represent aliasing. If an abstract term, abstract atom or abstract conjunction of atoms contains a_i several times (with the same subscript), the denoted concrete terms, atoms or conjunctions of atoms contain the *same* term in all positions corresponding to those occupied by a_i. For instance, the abstract conjunction $perm(g, a_1), ord(a_1)$ denotes the concrete conjunctions $\{perm(t_1, t_2), ord(t_2) | t_1, t_2 \in Term_p \text{ and } t_1 \text{ is ground}\}$.

In addition to g and a_i, we will include all concrete constants in the abstract domain, so $Con_p \subseteq ACon_p$. This is not essential for the approach: we could develop a sound and effective ACPD for the CC transformation based on the abstract constants g and $a_i, i \in \mathbb{N}$, alone. However, including Con_p in $ACon_p$ makes the analysis more precise: some redundant paths in the analysis are avoided.

Definition 1 (Abstract domain).
The abstract domain consists of:
– $ACon_p = Con_p \cup \{g\} \cup \{a_i | i \in \mathbb{N}\}$.
– $ATerm_p$, $AAtom_p$ *and* $AConAtom_p$ *are defined as the sets of the terms, atoms and conjunctions of atoms constructable from* $ACon_p$, Fun_p *and* $Pred_p$.

Next, we define the semantics of the abstract domain, through a concretization function γ. With slight abuse of notation, we use the same symbol γ to denote the concretization functions on $ATerm_p$, $AAtom_p$ and $AConAtom_p$.

In order to formalize the semantics of the aliasing, we need two auxiliary concepts: the subterm selection sequence and the aliasing context.

Definition 2 (Subterm selection sequence).
Let t be a term, atom or conjunction of atoms (either concrete or abstract).

– $i \in \mathbb{N}_0$ *is a subterm selection sequence for t, if $t = f(t_1, ..., t_n)$ and $i \leq n$. The subterm of t selected by i is t_i.*

– *$i_1.i_2.....i_n$ is a subterm selection sequence for t, if $t = f(t_1, ..., t_n)$, $i_1 \leq n$, $i_1 \in \mathbb{N}_0$ and $i_2.....i_n$ is a subterm selection sequence for t_{i_1}. With an inductively defined notation, we denote by $t_{i_1.i_2.....i_k}$ the subterm of $t_{i_1....i_{k-1}}$ selected by i_k, with $1 < k \leq n$. We also refer to $t_{i_1.i_2.....i_n}$ as the subterm of t selected by $i_1.i_2.....i_n$.*

Note that, in this definition, we assume that a conjunction of atoms A_1, A_2,...,A_n is denoted as $\wedge(A_1, A_2, ..., A_n)$.

Example 2 (Subterm selection sequence). Let $t = f(g(h(X), 5), f(h(a), Y))$, then $t_{1.1.1} = X$, $t_{2.1.1} = a$.

Definition 3 (Aliasing context).
Let t be an abstract term, atom or conjunction of atoms. The aliasing context of t, denoted $AC(t)$, is the finite set of pairs (sss_1, sss_2) of subterm selection sequences of t, such that $t_{sss_1} = t_{sss_2} = a_i$ for some $i \in \mathbb{N}$.

Example 3 (Aliasing context).
Let $t = p(f(a_2, g), a_1, a_2, g(h(a_1)))$, then $AC(t) = \{(1.1, 3), (2, 4.1.1)\}$.

Definition 4 (Concretization function).
The concretization function $\gamma : ATerm_p \cup AAtom_p \cup AConAtom_p \rightarrow 2^{Term_p} \cup 2^{Atom_p} \cup 2^{ConAtom_p}$ is defined as:

– *$\gamma(c) = \{c\}$, for any $c \in Con_p$*
– *$\gamma(g) = \{t \in Term_p | t$ is ground$\}$*
– *$\gamma(a_i) = Term_p$, $i \in \mathbb{N}$*
– *$\gamma(f(at_1, ..., at_n)) = \{f(t_1, ..., t_n) | t_i \in \gamma(at_i), i = 1...n$, and let t denote $f(t_1, ..., t_n)$, then for all $(sss_1, sss_2) \in AC(f(at_1, ..., at_n)) : t_{sss_1} = t_{sss_2}\}$*

Example 4 (Concretization function).
$\gamma(p(f(a_2, g), a_1, a_2, q(h(a_1)))) = \{p(f(t_1, t_2), t_3, t_1, q(h(t_3))) | t_1, t_3 \in Term_p, t_2$ ground term of $Term_p\}$

The abstract domain introduced above is infinitely large. There are two causes for this. Terms can be nested unboundedly deep, therefore infinitely many different terms exist. In addition, there are infinitely many $a_i, i \in \mathbb{N}$, symbols.

If so desired, the abstract domain can be refined, so that it becomes finite. This is done by using depth-k abstraction and by defining an equivalence relation on $\{a_i | i \in \mathbb{N}\}$. For the purpose of this paper, the infinite size of the abstract domain is not a problem.

Let us return to the permutation sort example. ACPD requires a top-level abstract atom (or conjunction) to start the transformation. Let $sort(g, a_1)$ be this atom. In the context of the $\mathcal{A}$-coveredness condition of partial deduction, our initial set $\mathcal{A}$ is $\{sort(g, a_1)\}$.

Below, we construct a finite number of finite, abstract partial deduction derivation trees for abstract (conjunctions of) atoms. The construction of these trees assumes an "abstract unification" and an "abstract unfold" operation. Their

formal definitions can be found in Annex [2014]. For now, we only show their effects in abstract partial derivation trees.

Next, we need an "oracle" that decides on the selection rule applied in the abstract derivation trees. This oracle mainly has two functions:

- to decide whether an obtained goal should be unfolded further, or whether it should be kept residual (to be split and added to $\mathcal{A}$),
- to decide which atom of the current goal should be selected for unfolding.

In fact, we will use a third type of decision that the oracle may make: it may decide to "fully evaluate" a selected atom. This type of decision is not commonly supported in partial deduction. What it means is that we decide not to transform a certain predicate of the original program, but merely keep its original definition in the transformed program. In partial deduction, this can be done by never selecting these atoms, including them in $\mathcal{A}$ and including their original definition in the transformed program.

In our setting, however, we want to know the effect that solving the atom has on the remainder of the goal. Therefore, we will assume that a full abstract interpretation over our abstract domain computes the abstract bindings that solving the atom results in. These are applied to the remainder of the goal. Note that this cannot easily be done in standard partial deduction, as fully evaluating an atom during (concrete) partial deduction may not terminate. In Vidal [2011], a similar functionality is integrated in a hybrid approach to conjunctive partial deduction.

For now, we simply assume the existence of the oracle. Fig. 1, 2, 3 show the abstract partial derivation trees that ACPD may build for permutation sort and top level $\mathcal{A} = \{sort(g, a_1)\}$.

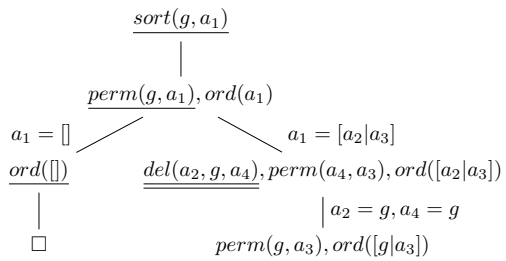

Fig. 1: Abstract tree for $sort(g, a_1)$

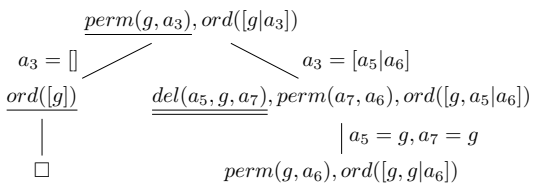

Fig. 2: Abstract tree for $perm(g, a_3), ord([g|a_3])$

In these figures, in each goal, the atom selected for abstract unfolding is underlined. If an atom is underlined twice, this expresses that the atom was selected for full abstract interpretation.

Both unfolding and full abstract evaluation may create bindings. Our abstract unification only collects bindings made on the a_i terms. Bindings created on g terms are not relevant.

In the left branch of the tree in Fig. 1 we see the effect of including the concrete constants in the abstract domain. As a result, the binding for a_1 is [], instead of g. If we had not included Con_p in $ACon_p$, then $ord(g)$ would have required a full analysis, using the three clauses for $ord/1$.

A goal with no underlined atom indicates that the oracle selects no atom and decides to keep the conjunction residual. After the construction of the tree in Fig. 1, ACPD adds the abstract conjunction $perm(g, a_3), ord([g|a_3])$ to $\mathcal{A}$. ACPD starts a new tree for this atom. This tree is shown in Fig. 2.

The tree is quite similar to the one in Fig. 1. The main difference is that, in the residual leaf, the ord atom now has a list argument with two g elements. This pattern does not yet exist in the current $\mathcal{A}$ and is therefore added to $\mathcal{A}$. A third abstract tree is computed for $perm(g, a_6), ord([g, g|a_6])$, shown in Fig. 3.

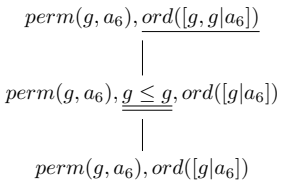

Fig. 3: Abstract tree for $perm(g, a_6), ord([g, g|a_6])$

In Fig. 3, the residual leaf $perm(g, a_6), ord([g|a_6])$ is a renaming of the conjunction $perm(g, a_3), ord([g|a_3])$, which is already contained in $\mathcal{A}$. Therefore, ACPD terminates the analysis, concluding $\mathcal{A}$-coveredness for $\mathcal{A} = \{sort(g, a_1), \wedge(perm(g, a_3), ord([g|a_3])), \wedge(perm(g, a_6), ord([g, g|a_6]))\}$.

In standard (concrete) conjunctive partial deduction, the analysis phase would now be completed. In ACPD, however, we need an additional step. In the abstract partial derivation trees, we have not collected the concrete bindings that unfolding would produce. These are required to generate the resolvents. Therefore, we need an additional step, constructing essentially the same three trees again, but now using concrete terms and concrete unification.

We only show one of these concrete derivation trees in Fig. 4. It corresponds to the tree in Fig. 2. We define the root of a concrete derivation tree corresponding to an abstract tree as follows.

Definition 5 (Concrete conjunctions in the root).
Let $acon \in \mathcal{A}$, then the conjunction in the root of the corresponding concrete tree, denoted as $c(acon)$, is obtained by replacing any g or a_i symbol in $acon$ by a fresh free variable, ensuring that multiple occurrences of a_i, with the same subscript i, are replaced by identical variables.

When unfolding the concrete tree, every abstract unfolding of the abstract tree is mimicked, using the same clauses, over the concrete domain.

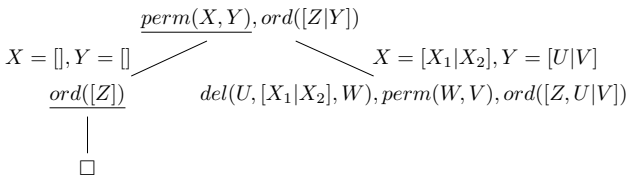

Fig. 4: Concrete tree corresponding to Fig. 2

The step of full abstract interpretation of the $del(a_5, g, a_7)$ atom in Fig. 2 has no counterpart in Fig. 4. The atom $del(U, [X_1|X_2], W)$ is kept residual and the $del/3$ clauses are added to the transformed program.

More specifically, using a renaming $p_1(X, Y, Z)$ for $\wedge(perm(X, Y), ord([Z|Y]))$ and $p_2(W, V, Z, U)$ for $\wedge(perm(W, V), ord([Z, U|V]))$, we synthesize the following resolvents from the tree in Fig. 4:

$p_1([], [], Z) \leftarrow .$
$p_1([X_1|X_2], [U|V], Z) \leftarrow del(U, [X_1|X_2], W), p_2(W, V, Z, U).$

From the counterparts of the trees in Fig. 1 and Fig. 3, we obtain the following additional resultants:

$sort([], []).$
$sort([X_1|X_2], [Y_1|Y_2]) \leftarrow del(Y_1, [X_1|X_2], Z), p_1(Z, Y_2, Y_1).$
$p_2(U, V, W, X) \leftarrow W \leq X, p_1(U, V, X).$

4 A More Complex Example, Introducing the *multi* Abstraction

In Section 3 we have shown that ACPD is indeed sufficient to formally revisit CC for a simple example. However, for more complex examples, ACPD still lacks expressivity. Consider the following map coloring program.

Example 5 (Map coloring).

```
safe_coloring(Ns,Cs) ← coloring(Cs),safe(Ns,Cs).
coloring([]).
coloring([C|Cs]) ← color(C),coloring(Cs).
safe([],[]).
safe([N],[C]).
safe([N1,N2|Ns],[C1,C2|Cs]) ←
    allsafe(N1,C1,[N2|Ns],[C2|Cs]), safe([N3|Ns],[C2|Cs]).
allsafe(N,C,[],[]).
allsafe(N1,C1,[N2|Ns],[C2|Cs]) ←
    test(N1,C1,N2,C2), allsafe(N2,C2,Ns,Cs).
test(N1,C1,N2,C2) ← edge(N1,N2),C1 ≠ C2; noedge(N1,N2).
```

In addition, we assume a number of facts for $edge/2$, $noedge/2$ and $color/1$, specifying the connections between a number of nations ($edge(n_1, n_2)$ and $noedge(n_1, n_2)$ facts) and a number of colors ($color(c)$ facts). The $allsafe/4$ predicate tests one nation with respect to all others. The $safe/2$ predicate tests all nations with respect to one another.

The program is intended to be called with a goal $safe\_coloring(Ns, Cs)$, with Ns a ground list of nations without duplicate entries and Cs a list of free variables.

The complete ACPD style analysis is available in Annex [2014]. We only present some relevant parts.

The top level goal for the abstract analysis is $safe\_coloring(g, a_1)$, so that the initial set $\mathcal{A} = \{safe\_coloring(g, a_1)\}$. A first abstract derivation tree describes the initialization for the computation. It contains two branches leading to an empty goal (success branches) and a third branch with the leaf: $\wedge(coloring([a_4|a_5]), allsafe(g, g, g, [a_4|a_5]), safe(g, [a_4|a_5]))$, which is added to $\mathcal{A}$.

Next, we construct an abstract derivation tree for the latter conjunction. This, again, gives a successful branch, with an empty conjunction in the leaf and a second branch, with the following leaf, which should be added to $\mathcal{A}$: $\wedge(coloring([a_6|a_7]), allsafe(g, g, g, [a_6|a_7]), allsafe(g, g, g, [a_6|a_7]), safe(g, [a_6|a_7]))$.

At this point it becomes clear that an analysis following only the steps shown in Section 3 will not terminate. The two abstract conjunctions, most recently added to $\mathcal{A}$, are identical – up to renaming of a_i's – except that the latter conjunction contains two atoms $allsafe(g, g, g, [a_i|a_j])$, instead of just one. A further analysis, building additional derivation trees, will result in the construction of continuously growing conjunctions, with continuously increasing numbers of $allsafe/4$ atoms.

We could solve this by cutting the goal into two smaller conjunctions and adding these to $\mathcal{A}$. However, all these atoms are generators or testers in the coroutine and depend on eachother. So, instead of splitting, we extend ACPD. One of the restrictions imposed by ACPD is that for any abstract conjunction of atoms, $acon \in AConAtom_p$, there exists a concrete conjunction, $con \in ConAtom_p$, such that: for all $con_i \in \gamma(acon)$: con_i is an instance of con. In practice, this means that an abstract conjunction is not allowed to represent a set of concrete conjunctions whose elements have a distinct number of conjuncts. However, in order to solve the problem observed in our example, we need the ability to represent a set of conjunctions, with a growing number of atoms, by an abstract atom.

We propose to introduce a new abstraction into our abstract domain: the $multi/1$ abstraction.

Definition 6 ($multi/1$ abstraction).
We extend $AConAtom_p$ with a new construct $multi(A)$, where $A \in AAtom_p$, such that: $\gamma(multi(A)) = \{\gamma(\bigwedge_{i=1\ldots n} A) | n \in \mathbb{N}_0\}$.

Example 6 ($multi/1$ abstraction).
The abstract conjunction
$multi(allsafe(g, g, g, [a_6|a_7]))$ denotes all finite concrete conjunctions $AS_1 \wedge \ldots \wedge AS_n, n \in \mathbb{N}_0, AS_i \in \gamma(allsafe(g, g, g, [a_6|a_7]))$ for each $i = 1, n$.

We now define two new operations on our abstract domain: abstract unfolding of $multi/1$ and abstract generalization with $multi/1$.

Definition 7 (Abstract unfolding of $multi/1$).
Let $A \in AAtom_p$. Abstract unfolding of $multi(A)$ is defined as standard unfolding using the clauses:
$multi(X) \leftarrow X.$
$multi(X) \leftarrow X, multi(X).$

The definition states that unfolding an abstract $multi(A)$ conjunction makes a case split: the case in which there is only one atom A in the abstract conjunction and the case in which there are more than one.

Definition 8 ($multi/1$ generalization).
Let $A \in AAtom_p$ and $n \in \mathbb{N}_0$. Abstract generalization with $multi/1$ is defined as the function $Gen : AConAtom_p \rightarrow AConAtom_p$:
$Gen(\bigwedge_{i=1\ldots n} A) = multi(A)$
$Gen((\bigwedge_{i=1\ldots n} A) \wedge multi(A)) = multi(A)$
$Gen(multi(A) \wedge (\bigwedge_{i=1\ldots n} A)) = multi(A)$

We illustrate these operations in our running example. Observing the growing number of $allsafe(g, g, g, [a_6|a_7])$ atoms in our last conjunction (w.r.t. the conjunction already present in $\mathcal{A}$), we perform the generalization: $Gen(\wedge(allsafe(g, g, g, [a_6|a_7]), allsafe(g, g, g, [a_6|a_7]))) = multi(allsafe(g, g, g, [a_6|a_7]))$

We replace the conjunction $\wedge(coloring([a_4|a_5]), allsafe(g, g, g, [a_4|a_5]), safe(g, [a_4|a_5]))$ from $\mathcal{A}$ by:
$\wedge(coloring([a_6|a_7]), multi(allsafe(g, g, g, [a_6|a_7])), safe(g, [a_6|a_7]))$

Then we construct a new abstract derivation tree for this conjunction, including – among other – an abstract unfold of $multi/1$ and abstract generalizations with $multi/1$. In Fig. 5, 6, 7, we show this abstract tree.

After abstract unfolding of $coloring([a_1|a_2])$ and full abstract evaluation of $color(a_1)$, the tree contains an abstract unfolding of $multi(allsafe(g, g, g, [g|a_2]))$. Fig. 6 contains the base case for this unfolding. After an unfold of $allsafe/4$ and full solve of $test/4$, unfolding of $safe(g, [g|a_2])$, using the clause $safe([N], [C]) \leftarrow .$, eventually leads to the empty goal. Unfolding $safe(g, [g|a_2])$ with the recursive clause produces a goal with two $allsafe(g, g, g, [a_3|a_4])$ atoms. We perform generalization with $multi/1$.

In Fig. 7, the non-base case for the abstract unfolding of $multi/1$ unfolds an $allsafe(g, g, g, [g|a_2])$ and fully solves a $test(g, g, g, g)$. Finally, we perform a generalization with $multi/1$ for the $allsafe(g, g, g, a_2)$ atom, reaching the new conjunction of $\mathcal{A}$: $\wedge(coloring(a_2), multi(allsafe(g, g, g, a_2)), multi(allsafe(g, g, g, [g|a_2])), safe(g, [g|a_2]))$

Eventually, the analysis ends up with a final set $\mathcal{A}$:

$$\{ \wedge(safe\_coloring(g, a_1)), \wedge(multi(allsafe(g, g, g, []))),$$
$$\wedge(coloring([a_1|a_2]), multi(allsafe(g, g, g, [a_1|a_2])), safe(g, [a_1|a_2])),$$
$$\wedge(coloring(a_2), multi(allsafe(g, g, g, a_2)), multi(allsafe(g, g, g, [g|a_2])), safe(g, [g|a_2]))\}$$

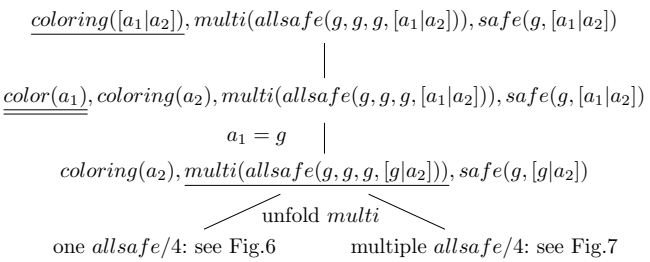

Fig. 5: A recurrent pattern in which a *multi* abstraction is unfolded by case analysis

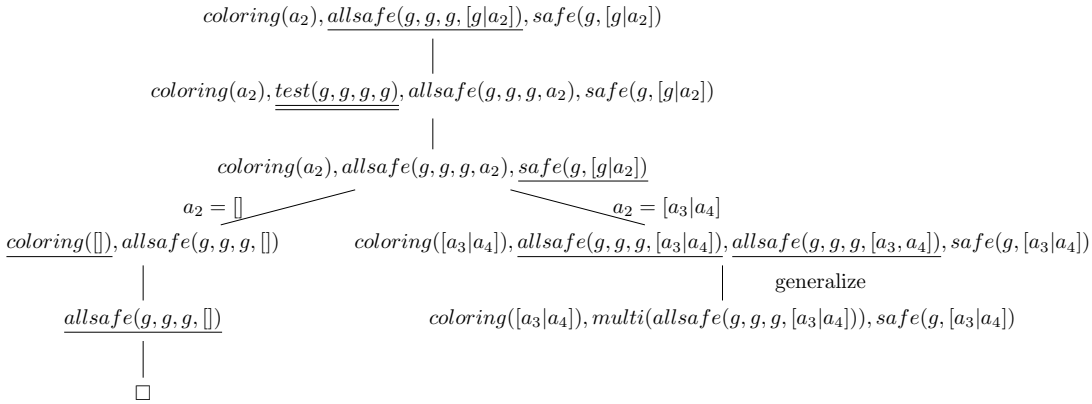

Fig. 6: First case from Fig. 5

All non-empty leaves in the abstract derivation trees for these atoms are (renamings of) elements of $\mathcal{A}$. This shows $\mathcal{A}$-coveredness and the abstract phase of the analysis terminates.

Similar to what was observed for permutation sort in Section 3, we still need an extra analysis to collect the concrete bindings, so that the resultants can be generated. Special care is required for the $multi/1$ abstraction. There are three issues: how to represent $multi/1$ in the concrete domain, how to deal with the concrete counterparts of abstract generalization with $multi/1$ and abstract unfolding of $multi/1$.

Definition 5, in Section 3, defined the concrete counterparts of the conjunctions in $\mathcal{A}$. We extend it to $multi(A)$:

Definition 9 (Concrete conjunction for $multi(A)$).
Let $A \in AAtom_p$, then $c(multi(A)) = multi([c(A)|T])$, with T a fresh variable.

Example 7. $c(multi(allsafe(g,g,g,a_2))) = multi([allsafe(X,Y,U,V)|T])$

For the abstract generalization with $multi/1$, we define the concrete counterpart as follows.

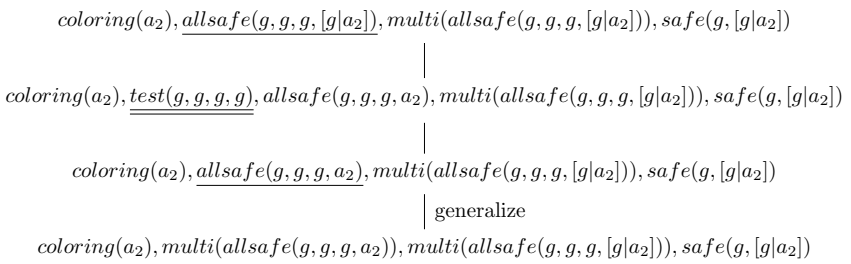

Fig. 7: Second case from Fig. 5

Definition 10 (Concrete generalization).
Let $A \in AAtom$.

- *If the abstract generalization with $multi/1$ is of the type $Gen(\bigwedge_{i=1,n} A) = multi(A)$, then the corresponding node in the concrete derivation contains $c(\bigwedge_{i=1,n} A)$. The concrete generalization is defined as $ConGen(c(\bigwedge_{i=1,n} A)) = multi(c([A,...,A]))$, with n members in the list.*
- *If the abstract generalization with $multi/1$ is of the type $Gen((\bigwedge_{i=1,n} A) \wedge multi(A)) = multi(A)$, then the corresponding node in the concrete derivation contains $c((\bigwedge_{i=1,n} A) \wedge multi(List))$, where $List$ is a list of at least one $c(A)$. The concrete generalization is defined as $ConGen(c((\bigwedge_{i=1,n} A) \wedge multi(List))) = multi(c([A,...,A|List]))$ with n new members added to $List$.*
- *The third case, $Gen(multi(A) \wedge (\bigwedge_{i=1,n} A)) = multi(A)$, is treated identically to the previous one.*

Example 8 (Concrete generalization).
Let $allsafe(X_1, Y_1, [U_1|V_1], [W|Z]), allsafe(X_2, Y_2, [U_2|V_2], [W|Z])$ occur in a concrete conjunction in a concrete derivation tree, where abstract generalization with $multi/1$ is performed on the corresponding abstract conjunction. Then, as a next step in the concrete derivation tree, this conjunction is replaced by $multi([allsafe(X_1, Y_1, [U_1|V_1], [W|Z]), allsafe(X_2, Y_2, [U_2|V_2], [W|Z])])$.

Note that this "generalization" actually does not generalize anything. It only brings the information in a form that can be generalized.

The actual generalization happens implicitly in the move to the construction of the next concrete derivation tree. If our conjunction is a leaf of the concrete derivation tree, then the corresponding abstract conjunction is added to the set $\mathcal{A}$. Let $\wedge(coloring([a_1|a_2]), multi(allsafe(g,g,g,[a_1|a_2])), safe(g,[a_1|a_2]))$, for instance, be the corresponding abstract conjunction that is added to $\mathcal{A}$. Then, a new concrete tree is built for a concrete atom corresponding to this abstract one.

In this example, the root of that concrete tree is:

$$\wedge(coloring([W|Z]), multi([allsafe(X,Y,V,[W|Z])|Tail]), safe(U,[W|Z]))$$

Finally, we still need to define the counterpart of abstract unfold of $multi/1$ in the concrete tree. To do this, we add the following definition of $multi/1$ to the original program P:

```
multi([X]) ← X.
multi([X|Y]) ← X,multi(Y).
```

It should be clear that mere concrete unfolding of concrete *multi*/1 atoms with the above definition for *multi*/1 gives us the desired counterpart of the case split performed in abstract unfold of *multi*/1.

With the concepts above, we construct a concrete derivation tree, mimicking the steps in the abstract derivation tree – but over the concrete domain – for every conjunction in the set $\mathcal{A}$. Collecting all the resultants from these concrete trees, we get the transformed program. A working Prolog program can be found in Annex [2014].

5 Discussion

In this paper, we have presented an approach to formally analyze the computations, for Logic Programs, performed under coroutining selection rules, and to compile such computations into new programs. On the basis of an example, we have shown that simple coroutines, in which the execution of a single, atomic generator is interleaved with a single, atomic tester, can be successfully analyzed and compiled within the framework of ACPD (Leuschel [2004]). These "simple" coroutines essentially correspond to the *strongly regular* logic programs of Vidal [2011].

To achieve this, we defined an expressive abstract domain, capturing modes, types and aliasing. In the paper, we have focused on the intuitions, more than on the full formalization, as space restrictions would not allow both. However, we have developed the formal definitions for the ordering on the abstract domain, abstract unification, abstract unfold and others.

Because the approach – for simple coroutines – fits fully within the ACDP framework, it inherits the correctness results from ACPD. In particular, $\mathcal{A}$-closedness and independence guarantee the completeness and correctness of the analysis. In addition, the transformation preserves all computed answers (in both directions) and finite failure of the transformed program implies finite failure of the original.

We have proposed an extension to our abstract domain: the *multi*/1-abstraction. A *multi*/1 atom can represent (sets of) conjunctions of one or more concrete atoms. We have defined abstract unfold and abstract generalisation operations for this abstraction. We have shown, in an example, that this abstraction and these operations allow us to extend ACPD, enabling it to perform a complete analysis, and to compile the more complex coroutines.

On a more general level, our work provides a new, rational reconstruction of the CC-transformation (Bruynooghe et al. [1986]), avoiding ad hoc features of the CC approach. In addition, the work presents a new application for ACPD.

As a rule, coroutining improves the efficiency of declarative programs by testing partial solutions as quickly as possible. In addition, a program may become more flexible when the transformation is applied. Recall that the original is meant to be called with a ground list of nations and a list of free variables. The

transformed program can be run in the same way, but the top-level predicate can also be called with a ground list of nations and a free variable. This is because SLD resolution sends the original program down an infinite branch of the search tree. The transformed program checks results earlier and, as a result, infers that both top-level arguments must be lists of the same size. In this scenario, compiling control transforms an infinite computation into a finite one.

The CC-transformation raised challenges for a number of researchers and a range of compediting transformation and synthesis techniques. A first reformulation of the CC-transformation was proposed in the context of the "programs-as-proofs" paradigm, in Wiggins [1990]. It was shown that CC-transformations, to a limited extent, could be formalized in a proof-theoretic program synthesis context.

In Boulanger et al. [1993], CC-transformation was revisited on the basis of a combination of abstract interpretation and constraint processing. This improved the formalization of the technique, but it did not clarify the relation with partial deduction.

The seminal survey paper on Unfold/Fold transformation, Pettorossi and Proietti [1994], showed that basic CC-transformations are well in the scope of Unfold/Fold transformation. In later works (e.g. Pettorossi and Proietti [2002]), the same authors introduced list-introduction into the Unfold/Fold framework, whose function is very similar to that of the *multi*/1 abstraction in our approach. Also related to our work are Puebla et al. [1997], providing alternative transformations to improve the efficiency of dynamic scheduling, and Vidal [2011] and Vidal [2012], which also provide a hybrid form of partial deduction, combining abstract and concrete levels.

There are a number of issues that are open for future research. First, we aim to investigate the generality of the *multi*/1 abstraction. Although it seems to work well in a number of examples, we will study more complex ones. We also want to revisit the ACPD framework, in order to extend it to the new abstraction we aim to support. This will involve a new formalization of ACPD, capable of supporting analysis and compilation of coroutines in full generality. This will also formally establish the correctness results for the more general cases, such as the one presented in Section 4. Obviously, we also want to have a full implementation of these concepts and to show that the analysis and compilation can be fully automated.

6 Acknowledgements

We thank the anonymous reviewers for their very useful suggestions.

References

Annex. Definitions, concepts and elaboration of an example. `http://perswww.kuleuven.be/~u0055408/appendices-acpd.html`, 2014.

Dmitri Boulanger, Maurice Bruynooghe, and Danny De Schreye. Compiling control revisited: A new approach based upon abstract interpretation for constraint logic programs. In *LPE*, pages 39–51, 1993.

Maurice Bruynooghe. A practical framework for the abstract interpretation of logic programs. *The Journal of Logic Programming*, 10(2):91–124, 1991.

Maurice Bruynooghe, Danny De Schreye, and Bruno Krekels. Compiling control. In *Proceedings of the 1986 Symposium on Logic Programming*, Salt Lake City, 1986. IEEE Society Press.

Maurice Bruynooghe, Danny De Schreye, and Bruno Krekels. Compiling control. *The Journal of Logic Programming*, 6(1):135–162, 1989.

Danny De Schreye, Robert Glück, Jesper Jørgensen, Michael Leuschel, Bern Martens, and Morten Heine Sørensen. Conjunctive partial deduction: Foundations, control, algorithms, and experiments. *The Journal of Logic Programming*, 41(2):231–277, 1999.

John P. Gallagher. Transforming logic programs by specialising interpreters. In *ECAI*, pages 313–326, 1986.

Michael Leuschel. A framework for the integration of partial evaluation and abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(3):413–463, 2004.

John Lloyd. *Foundations of Logic Programming*. Berlin: Springer-Verlag, 1987.

John W. Lloyd and John C Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11(3):217–242, 1991.

Alberto Pettorossi and Maurizio Proietti. Transformation of logic programs: Foundations and techniques. *The Journal of Logic Programming*, 19:261–320, 1994.

Alberto Pettorossi and Maurizio Proietti. The list introduction strategy for the derivation of logic programs. *Formal aspects of computing*, 13(3-5):233–251, 2002.

Germán Puebla, Maria J Garcia de la Banda, Kim Marriott, and Peter J Stuckey. Optimization of logic programs with dynamic scheduling. In *ICLP*, volume 97, pages 93–107, 1997.

Germán Vidal. *A hybrid approach to conjunctive partial evaluation of logic programs*. Springer, 2011.

Germán Vidal. Annotation of logic programs for independent and-parallelism by partial evaluation. *Theory and Practice of Logic Programming*, 12(4-5):583–600, 2012.

Geraint A Wiggins. *The improvement of prolog program efficiency by compiling control: A proof-theoretic view*. Department of Artificial Intelligence, University of Edinburgh, 1990.